Lineare Algebra

Endliche Arithmetik

Walter Gander

ETH Zürich

Contents

Chapter 1. Finite Arithmetic **1**
1.1 Introductory Example 1
1.2 Real Numbers and Machine Numbers 2
1.3 The IEEE Standard 5
1.3.1 Single Precision 5
1.3.2 Double Precision 7
1.4 Computations with Machine Numbers 9
1.4.1 Rounding Errors 9
1.4.2 Associative Law 9
1.4.3 Summation Algorithm by W. Kahan 11
1.4.4 Small Numbers 11
1.4.5 Monotonicity 11
1.4.6 Avoiding Overflow 12
1.4.7 Test for Overflow 14
1.4.8 Cancellation 14
1.5 Machine-independent Algorithms 21
1.6 Termination Criteria 23
1.6.1 Test Successive Approximations 24
1.6.2 Check Residual 24
1.7 Condition and Stability 25
1.8 Principle of Wilkinson 25
1.9 The Condition of a System of Linear Equations 26
1.10 Stable and Unstable Algorithms 26

Chapter 1. Finite Arithmetic

1.1 Introductory Example

A very old problem already studied by ancient Greek mathematicians is the *squaring of a circle*. The problem consists in transforming a circle into a coextensive square by using straight edge and compass only. To transform this way a circle in a square (*quadrature of the circle*) became a famous unsolved problem for centuries until it was proved by Galois theory in the 19th century that the problem cannot be solved using straight edge and compass.

We know today that the circle area is given by $A = r^2\pi$, where r denotes the radius of the circle. An approximation is obtained by drawing a regular polygon inside the circle and by computing the surface of the polygon. The approximation is improved by increasing the number of corners.

Archimedes managed to produce a 96-sided polygon and by this was able to enclose π in the interval $(3\frac{1}{7}, 3\frac{10}{71})$. The enclosing interval has length $1/497 = 0.00201207243$ — surely good enough for most practical applications.

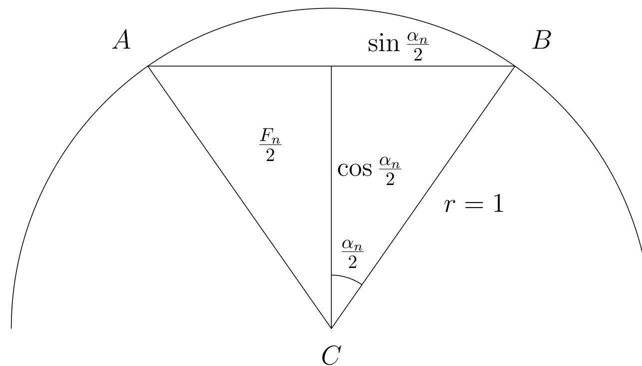

FIGURE 1.1. *Squaring of a Circle*

To compute such a polygonal approximation of π we consider Figure 1.1. Without loss of generality we may assume that $r = 1$. Then the area F_n of the isosceles triangle ABC with center angle α_n is

$$F_n = \cos\frac{\alpha_n}{2}\sin\frac{\alpha_n}{2}$$

and the area of the associated n-sided polygon becomes

$$A_n = nF_n = \frac{n}{2}\left(2\cos\frac{\alpha_n}{2}\sin\frac{\alpha_n}{2}\right) = \frac{n}{2}\sin\alpha_n = \frac{n}{2}\sin\left(\frac{2\pi}{n}\right).$$

Clearly, computing the approximation A_n using π would be rather contradictory. Fortunatly A_{2n} can be derived from A_n by simple algebraic transformations, i.e. by expressing $\sin(\alpha_n/2)$ in terms of $\sin\alpha_n$.

This can be achieved by using identities for trigonometric functions:

$$\sin\frac{\alpha_n}{2} = \sqrt{\frac{1-\cos\alpha_n}{2}} = \sqrt{\frac{1-\sqrt{1-\sin^2\alpha_n}}{2}}. \tag{1.1}$$

Thus we have obtained a recursion for $\sin(\alpha_n/2)$ from $\sin\alpha_n$. To start the recursion we compute the area A_6 of the regular hexagon. Each of the six equilateral triangles has side length 1. The angle $\alpha_6 = 60°$ and hence $\sin\alpha_6 = \frac{\sqrt{3}}{2}$. Therefore, the area of the triangle is $F_6 = \sqrt{3}/4$ and $A_6 = 3\frac{\sqrt{3}}{2}$. We obtain the following program to compute the sequence of approximations A_n:

ALGORITHM 1.1. *Computation of π, Naive Version*

```
% computation of pi, naive version
s = sqrt(3)/2; A=3*s; n=6;    % initialisation
z =  [A-pi n  A  s];          % store the results
while (s > 1e-10)             % termination if s = sin(alpa) small
  s= sqrt((1-sqrt(1-s*s))/2); % new sin(alpha/2) value
  n=2*n; A= n/2*s;            % A = new polygon area
  z = [z; A-pi n  A  s];
end
m = length(z);
for i=1:m
   fprintf('%10d %20.15f %20.15f  %20.15f\n', z(i,2),z(i,3),...
       z(i,1),z(i,4))
end
```

The results, displayed in Table 1.1, are not what we would expect: initially we observe convergence towards π, for $n > 49152$, the error grows again and finally $A_n = 0$? *Though the theory and the program are correct, we obtain wrong answers.* We will explain in this chapter why this is the case.

1.2 Real Numbers and Machine Numbers

Every computer is a finite automaton. This implies that a computer can only store a finite set of numbers and perform only a finite number of operations. In mathematics we are used to carry out our computations with real numbers $\mathbb{R}$ covering the continuous interval $(-\infty,\infty)$. On the computer we deal with a discrete finite set of machine numbers $\mathbb{M} = \{-\tilde{a}_{min},\ldots,\tilde{a}_{max}\}$. Hence each real number a has to be mapped onto a machine number $\tilde{a}$ to be used on a computer. In fact a whole interval of real numbers is mapped onto one machine number as shown in Figure 1.2. Nowadays machine numbers are often represented in the *binary system.* In general any base (or *radix*) B could be used to represent numbers. A real machine number or *floating point number* consists of two parts, a *mantissa* (or *significand)* m and an *exponent* e

$$\begin{array}{rcll} \tilde{a} &=& \pm m \times B^e & \\ m &=& D.D\cdots D & \text{mantissa} \\ e &=& D\cdots D & \text{exponent} \end{array}$$

n	A_n	$A_n - \pi$	$sin(\alpha_n)$
6	2.598076211353316	−0.543516442236477	0.866025403784439
12	3.000000000000000	−0.141592653589794	0.500000000000000
24	3.105828541230250	−0.035764112359543	0.258819045102521
48	3.132628613281237	−0.008964040308556	0.130526192220052
96	3.139350203046872	−0.002242450542921	0.065403129230143
192	3.141031950890530	−0.000560702699263	0.032719082821776
384	3.141452472285344	−0.000140181304449	0.016361731626486
768	3.141557607911622	−0.000035045678171	0.008181139603937
1536	3.141583892148936	−0.000008761440857	0.004090604026236
3072	3.141590463236762	−0.000002190353031	0.002045306291170
6144	3.141592106043048	−0.000000547546745	0.001022653680353
12288	3.141592516588155	−0.000000137001638	0.000511326906997
24576	3.141592618640789	−0.000000034949004	0.000255663461803
49152	3.141592645321216	−0.000000008268577	0.000127831731987
98304	3.141592645321216	−0.000000008268577	0.000063915865994
196608	3.141592645321216	−0.000000008268577	0.000031957932997
393216	3.141592645321216	−0.000000008268577	0.000015978966498
786432	3.141592303811738	−0.000000349778055	0.000007989482381
1572864	3.141592303811738	−0.000000349778055	0.000003994741190
3145728	3.141586839655041	−0.000005813934752	0.000001997367121
6291456	3.141586839655041	−0.000005813934752	0.000000998683561
12582912	3.141674265021758	0.000081611431964	0.000000499355676
25165824	3.141674265021758	0.000081611431964	0.000000249677838
50331648	3.143072740170040	0.001480086580246	0.000000124894489
100663296	3.137475099502783	−0.004117554087010	0.000000062336030
201326592	3.181980515339464	0.040387861749671	0.000000031610136
402653184	3.000000000000000	−0.141592653589793	0.000000014901161
805306368	3.000000000000000	−0.141592653589793	0.000000007450581
1610612736	0.000000000000000	−3.141592653589793	0.000000000000000

TABLE 1.1. *Unstable Computation of π*

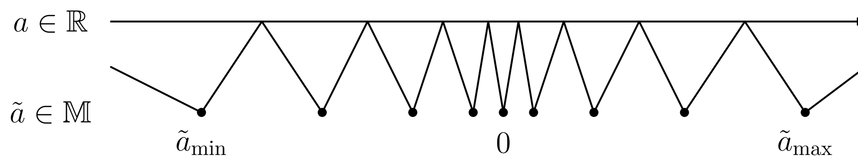

FIGURE 1.2.
Mapping of real numbers $\mathbb{R}$ onto machine numbers $\mathbb{M}$

where $D \in \{0, 1, \dots, B-1\}$ stands for one digit. To make the representation of machine numbers unique (note that e.g. $1.2345 \times 10^3 = 0.0012345 \times 10^6$), it is required that for a machine number $\tilde{a} \neq 0$ the first digit before the decimal point in the mantissa be nonzero. Such numbers are called *normalized*. Characteristic for such a *finite arithmetic* is the number of digits used for the mantissa and the exponent. The number of digits in the exponent defines the *range of the machine numbers*. The numbers of digits in the mantissa defines the precision.

More specifically [?], a finite arithmetic is defined by four integer numbers: B, the base or radix, p, the number of digits in the mantissa, and L and U defining the exponent range: $L \leq e \leq U$.

The *machine precision* is described by the real machine number ϵ_{mach}. Traditionally, ϵ_{mach}*is defined to be the smallest* $\tilde{a} \in \mathbb{M}$ *such that* $\tilde{a} + 1 \neq 1$ *when the addition is carried out on the computer*. Because this definition involves details about the behavior of floating point addition, which are not easily accessible, a newer definition of $\epsilon_{\text{mach}}$is simply *the spacing of the floating point numbers between 1 and 2*. The current definition only involves how the numbers are represented.

Simple *calculators* often use the familiar decimal system ($B = 10$). Typically there are $p = 10$ digits for the mantissa and 2 for the exponent ($L = -99$ and $U = 99$). In this finite arithmetic we have

- `eps` $= 0.000000001 = 1.000000000 \times 10^{-9}$
- the largest machine number

$$\tilde{a}_{max} = 9.999999999 \times 10^{+99}$$

- the smallest machine number

$$\tilde{a}_{min} = -9.999999999 \times 10^{+99}$$

- the smallest (normalized) positive machine number

$$\tilde{a}_{+} = 1.000000000 \times 10^{-99}$$

Early computers e.g. the MARK 1 designed by Howard Aiken and Grace Hopper in Harvard and finished 1944 or the ERMETH (Elektronische Rechenmaschine der ETH) constructed by Heinz Rutishauser, Ambros Speiser and Eduard Stiefel, were also decimal machines. The ERMETH finished 1956 was operational at ETH Zurich from 1956–1963. The representation of a real number used 16 decimal digits: The first digit, the q-digit, stored the sum of the digits modulo 3. This was used as a check if the machine word had been transmitted correctly from memory to the registers. The next three digits contained the exponent. Then the next 11 digits represented the mantissa and finally the last digit held the sign. The range of positive machine numbers was $1.0000000000 \times 10^{-200} \leq \tilde{a} \leq 9.9999999999 \times 10^{199}$. The possible larger exponent range was not fully used.

The very first free programmable computer built by the German civil engineer Konrad Zuse, namely the Z3 presented in 1941 to a group of experts only, however, was already using the binary system. The Z3 worked with an exponent of 7 bits and

a mantissa of 14 bits (actually 15, since the numbers were normalized). The range of positive machine numbers was the interval

$$[2^{-63},\ 1.11111111111111 \times 2^{62}] \approx [1.08 \times 10^{-19},\ 9.22 \times 10^{18}].$$

In MAPLE (a computer algebra system) numerical computations are performed in decimal. The number of digits of the mantissa is defined by the variable `Digits` which can be deliberately chosen. The number of digits of the exponent is given by the word length of the computer – for 32 bits we have a huge maximal exponent of $U = 2^{31} = 2147483648$.

1.3 The IEEE Standard

Since 1985 there exists for computer hardware the *ANSI/IEEE Standard 754 for Floating Point Numbers*. It has been adopted by almost all computer manufacturers. The base is $B = 2$.

1.3.1 Single Precision

The IEEE single precision floating point standard representation uses a 32 bit word with bits numbered from 0 to 31 from left to right. The first bit S is the sign bit, the next eight bits E are the exponent bits, $e = EEEEEEEE$, and the final 23 bits are the bits F of the mantissa m:

$$\begin{array}{cll} S & \overbrace{EEEEEEEE}^{e} & \overbrace{FFFFFFFFFFFFFFFFFFFFFFF}^{m} \\ 0 & 1 \qquad\qquad 8 & 9 \qquad\qquad\qquad\qquad\qquad 31 \end{array}$$

The value $\tilde{a}$ represented by the 32 bit word is defined as follows:

normal case: If $0 < e < 255$ then $\tilde{a} = (-1)^S \times 2^{e-127} \times 1.m$ where $1.m$ is the binary number created by prefixing m with an implicit leading 1 and a binary point.

exceptions: If $e = 255$ and $m \neq 0$, then $\tilde{a} = NaN$ (*Not a number*)

If $e = 255$ and $m = 0$ and $S = 1$, then $\tilde{a} = -Inf$

If $e = 255$ and $m = 0$ and $S = 0$, then $\tilde{a} = Inf$

special cases: If $e = 0$ and $m \neq 0$, then $\tilde{a} = (-1)^S \times 2^{-126} \times 0.m$ These are *denormalized (or subnormal) numbers.*

If $e = 0$ and $m = 0$ and $S = 1$, then $\tilde{a} = -0$

If $e = 0$ and $m = 0$ and $S = 0$, then $\tilde{a} = 0$

Some examples:

```
0 10000000 00000000000000000000000 = +1 x 2^(128-127) x 1.0   =  2
0 10000001 10100000000000000000000 = +1 x 2^(129-127) x 1.101 =  6.5
1 10000001 10100000000000000000000 = -1 x 2^(129-127) x 1.101 = -6.5

0 00000000 00000000000000000000000 =  0
1 00000000 00000000000000000000000 = -0
```

```
0 11111111 00000000000000000000000 =  Inf
1 11111111 00000000000000000000000 = -Inf

0 11111111 00000100000000000000000 = NaN
1 11111111 00100010001001010101010 = NaN

0 00000001 00000000000000000000000 = +1 x 2^(1-127) x 1.0 =  2^(-126)
0 00000000 10000000000000000000000 = +1 x 2^(-126) x 0.1  =  2^(-127)

0 00000000 00000000000000000000001
 =  +1 x 2^(-126) x 0.00000000000000000000001 = 2^(-149)
 =  smallest positive denormalized machine number
```

In MATLAB real numbers are usually represented in double precision. The function `single` can be used, however, to convert numbers to single precision. Thus we get with

```
>> format hex
>> x = single(2)
x =
   40000000
>> 2
ans =
   4000000000000000
>> s = realmin('single')*eps('single')
s =
00000001
>> format long
>> s
s =
   1.4012985e-45
>> s/2
ans =
     0
% Exceptions
>> z = sin(0)/sqrt(0)
Warning: Divide by zero.
z =
   NaN
>> y = log(0)
Warning: Log of zero.
y =
  -Inf
>> t = cot(0)
Warning: Divide by zero.
> In cot at 13
t =
   Inf
```

We can see that `x` represents the number 2 in single precision. The functions `realmin` and `eps` with parameter `'single'` compute the machine constants for single precision. This means that `s` is the smallest denormalized number in single precision. Dividing `s` by 2 we get zero because of underflow. The computation of `z` yields an undefined expression which results in `NaN` even though the limit is defined. The other two computations for `y` and `t` show the exceptions `Inf` and `-Inf`.

1.3.2 Double Precision

The IEEE double precision floating point standard representation uses a 64 bit word with bits numbered from 0 to 63 from left to right. The first bit S is the sign bit, the next eleven bits E are the exponent bits for e and the final 52 bits F represent the mantissa m:

$$\begin{array}{llll} S & \overbrace{EEEEEEEEEEE}^{e} & & \overbrace{FFFFF\cdots FFFFF}^{m} \\ 0 & 1 \qquad\qquad\qquad 11 & & 12 \qquad\qquad\qquad 63 \end{array}$$

The value $\tilde{a}$ represented by the 64 bit word is defined as follows:

normal case: If $0 < e < 2047$ then $\tilde{a} = (-1)^S \times 2^{e-1023} \times 1.m$ where $1.m$ is the binary number created by prefixing m with an implicit leading 1 and a binary point.

exceptions: If $e = 2047$ and $m \neq 0$, then $\tilde{a} = NaN$ (*Not a number*)

If $e = 2047$ and $m = 0$ and $S = 1$, then $\tilde{a} = -Inf$

If $e = 2047$ and $m = 0$ and $S = 0$, then $\tilde{a} = Inf$

special cases: If $e = 0$ and $m \neq 0$, then $\tilde{a} = (-1)^S \times 2^{-1022} \times 0.m$ These are *denormalized* numbers.

If $e = 0$ and $m = 0$ and $S = 1$, then $\tilde{a} = -0$

If $e = 0$ and $m = 0$ and $S = 0$, then $\tilde{a} = 0$

In MATLAB, real computations are performed in IEEE double precision by default. It is convenient to print real numbers using the hexadecimal format to see the internal representation, e.g.

```
>> format hex
>> 2
ans =   4000000000000000
```

If we expand each hexadecimal digit to 4 binary digits we obtain for the number 2:

```
0100 0000 0000 0000 0000 0000 ....  0000 0000 0000
```

We skipped with `....` seven times a group of four zero binary digits. The interpretation is: $+1 \times 2^{1024-1023} \times 1.0 = 2$.

```
>> 6.5
ans =   401a000000000000
```

This means

```
0100 0000 0001 1010 0000 0000 ....  0000 0000 0000
```

Again we skipped with `....` seven times a group of four zeros. The resulting number is $+1 \times 2^{1025-1023} \times (1 + \frac{1}{2} + \frac{1}{8}) = 6.5$

We will concentrate in the following discussion on double precision since this is today the normal computation mode for real numbers in the IEEE Standard. Furthermore we stick to the IEEE standard as used in MATLAB. In other more low level programming languages, the behavior of the IEEE arithmetic can be adapted, e.g. the exception handling can be explicitly specified.

- The *machine precision* is `eps`$= 2^{-52}$.

- The largest machine number $\tilde{a}_{max}$ is denoted by `realmax`. Note that

```
>> realmax
   ans = 1.7977e+308

>> log2(ans)
   ans = 1024

>> 2^1024
   ans = Inf
```

This looks first like a contradiction since the largest exponent should be according the IEEE conventions $2^{2046-1023} = 2^{1023}$. But `realmax` is the number with the largest possible exponent and with the mantissa F consisting of all 1:

```
>> format hex
>> realmax
ans =   7fefffffffffffff
```

This is

$$\begin{aligned} V &= +1 \times 2^{2046-1023} \times 1.\underbrace{11\ldots1}_{52\,\text{Bits}} \\ &= 2^{1023} \times \left(1 + \left(\frac{1}{2}\right)^1 + \left(\frac{1}{2}\right)^2 + \cdots + \left(\frac{1}{2}\right)^{52}\right) \\ &= 2^{1023} \times \frac{1 - \left(\frac{1}{2}\right)^{53}}{1 - \left(\frac{1}{2}\right)} = 2^{1023} \times (2 - \texttt{eps}) \end{aligned}$$

In spite of `log2(realmax)=1024` we have `realmax` $\neq 2^{1024}$ but rather $(2-eps) \times 2^{1023}$.

- The *computation range* is the interval [−`realmax`, `realmax`]. If an operation produces a result outside this interval then it is said to *overflow*. Before the IEEE Standard, computation would stop in such a case with an error message. Now the result of an overflow operation is assigned the number ±`Inf`.

- The smallest positive normalized number is `realmin` $= 2^{-1022}$.

- IEEE allows computations with denormalized numbers. The positive denormalized numbers are in the interval [`realmin` ∗ `eps`, `realmin`]. If an operation produces a positive number which is not zero but also smaller than `realmin`∗`eps` then this result is in the *underflow range*. Such a result cannot be represented and zero is assigned instead.

- When computing with denormalized numbers we may suffer a loss of precision. Consider the following MATLAB program

```
>> format long
>> res = pi*realmin/123456789101112
```

```
res =  5.681754927174335e-322

>> res2 = res*123456789101112/realmin

   res2 = 3.15248510554597

>> pi   = 3.14159265358979
```

The first result `res` is a denormalized number – it cannot be represented with full accuracy anymore. So if we reverse the operations and compute `res2` we obtain here a result which has only 2 correct decimal digits. We recommend therefore to avoid computing with denormalized numbers.

1.4 Computations with Machine Numbers

1.4.1 Rounding Errors

Let $\tilde{a}$ and $\tilde{b}$ be two machine numbers then $c = \tilde{a} \times \tilde{b}$ will in general not be a machine number anymore since the product of two numbers consists of the double amount of digits. The result will therefore be a machine number $\tilde{c}$ which is next to c.

As an example consider the 8 digits decimal numbers

$$\tilde{a} = 1.2345678 \quad \text{and} \quad \tilde{b} = 1.1111111$$

Their product is

$$c = 1.37174198628258 \quad \text{and} \quad \tilde{c} = 1.3717420.$$

The *absolute rounding error* is the difference $r_a = \tilde{c} - c = 1.371742e{-}8$ and

$$r = \frac{r_a}{c} = 1e{-}8$$

is the *relative rounding error*.

On todays computers the following holds:

$$a \tilde{\oplus} b = (a \oplus b)(1 + r)$$

where r is the relative rounding error with $|r| < \varepsilon =$ machine precision. We denote with $\oplus \in \{+, -, \times, /\}$ the exact basic operation and with $\tilde{\oplus}$ the equivalent computer operation.

1.4.2 Associative Law

Consider the associative law

$$(a + b) + c = a + (b + c).$$

It does not hold in finite arithmetic. As an example take the three numbers

$$a = 1.23456e{-}3, \quad b = 1.00000e0, \quad c = -b.$$

Then it is easy to see that in decimal arithmetic we obtain $(a+b)+c = 1.23000e{-3}$ but $a+(b+c) = a = 1.23456e{-3}$.

It is therefore important to use parenthesis wisely and also to consider the order of the operations.

Assume e.g. that we have to compute a sum $\sum_{i=1}^{N} a_i$ where the terms $a_i > 0$ are monotonically decreasing: $a_1 > a_2 > \cdots > a_n$. As an example consider the harmonic series

$$S = \sum_{i=1}^{N} \frac{1}{i}.$$

For $N = 10^6$ we compute with sufficient accuracy `Digits := 20` in MAPLE an "exact" reference value:

```
Digits := 20;

s := 0;
for i from 1 to 1000000 do
 s := s+1.0/i:
od:
s;
      14.392726722865723804
```

Using MATLAB with IEEE arithmetic we get

```
N = 1e6
format long e
s1 = 0
 for i=1:N
   s1 = s1+1/i;
 end
s1

ans =  1.439272672286478e+01
```

We observe that the last three digits are different from the MAPLE result. If we sum again with MATLAB but in reverse order we obtain

```
s2 = 0
 for i=N:-1:1
   s2 = s2+1/i;
 end
s2
ans = 1.439272672286575e+01
```

a much better result! It differs only in the last digit from the MAPLE result. If we add a small number to a large one then the lower part of the smaller machine number is lost. We saw this effect in the example for the associative law. Thus it is better to start with the smallest elements in a sum and add the largest elements last. It would pay therefore to first sort the terms of the sum. But this means more computational work.

1.4.3 Summation Algorithm by W. Kahan

An accurate algorithm that does not need to sort the terms was given by W. Kahan. The idea here is not to discard but keep as carry the lower part of the small term which is added to the partial sum. The carry is added to the next term.

ALGORITHM 1.2. *Kahan's Summation of* $\sum_{j=1}^{N} \frac{1}{j}$

```
s = 0;
c = 0;
 for j=1:N
   y = 1/j + c;
   t = s+y;
   c = (s-t)+y;
   s=t;
 end
 s = s+c
```

Doing so we get a remarkably good result which agrees to the last digit with the MAPLE result:

```
s =  1.439272672286572e+01
```

1.4.4 Small Numbers

If $a + x = a$ holds then we conclude in mathematics that $x = 0$. This is not true in finite arithmetic. In IEEE arithmetic e.g. $1 + 1e{-}20 = 1$ holds and this is true not only for $1e{-}20$ but for all positive machine numbers w with $w <$ `eps`, where `eps` is the machine precision.

1.4.5 Monotonicity

Assume we are given a function f which is strictly monotonically increasing in $[a, b]$. Then for $x_1 < x_2$ with $x_i \in [a, b]$ we have $f(x_1) < f(x_2)$. Take as an example $f(x) = \sin(x)$ and $0 < x_1 < x_2 < \frac{\pi}{2}$. Can we be sure that in finite arithmetic also $\sin(x_1) < \sin(x_2)$ holds? The general answer is no. Though for *standard functions* special care was taken when implementing those in IEEE arithmetic that monotonicity was maintained so that at least $\sin(x_1) \le \sin(x_2)$ holds.

As an example let us consider the polynomial

$$f(x) = x^3 - 3.000001x^2 + 3x - 0.999999.$$

This function is almost $(x - 1)^3$, but has 3 simple zeros which are close:

$$0.998586, \quad 1.00000, \quad 1.001414.$$

Let us plot the function f:

```
figure(1)
a = -1; b = 3; h = 0.1;
x = a:h:b; y = x.^3 -3.000001*x.^2 +3*x -0.999999;
```

```
plot(x,y)
hold on
legend('x^3 -3.000001*x^2 +3*x -0.999999')
plot([a,b],[0,0])

figure(2)
a = 0.998; b = 1.002; h = 0.0001;
x = a:h:b; y = x.^3 -3.000001*x.^2 +3*x -0.999999;
plot(x,y)
hold on
legend('x^3 -3.000001*x^2 +3*x -0.999999')
plot([a,b],[0,0])

figure(3)
a = 0.999999993; b = 1.000000007; h = 0.000000000005;
x = a:h:b; y = x.^3 -3.000001*x.^2 +3*x -0.999999;
axis([a b -1e-13 1e-13])
plot(x,y)
hold on
legend('x^3 -3.000001*x^2 +3*x -0.999999')
plot([a,b],[0,0])

figure(4)  %  using  Horner's rule
a = 0.999999993; b = 1.000000007; h = 0.000000000005;
x = a:h:b;  y = ((x -3.000001).*x +3).*x -0.999999;
axis([a b -1e-13 1e-13])
plot(x,y)
hold on
legend('((x -3.000001)*x +3)*x -0.999999')
plot([a,b],[0,0])
```

If we zoom in to the zero 1 we see in Figure 1.3 that f behaves like a step function and we cannot ensure monotonicity. The steps are less pronounced if we use for the evaluation Horner's rule.

1.4.6 Avoiding Overflow

To avoid overflow, it is often necessary to modify the way how quantities are computed. Assume e.g. we wish to compute the polar coordinates of a given point (x, y) in the plane. To compute the radius $r > 0$ the textbook approach is to use

$$r = \sqrt{x^2 + y^2}.$$

However, if $|x|$ or $|y|$ is larger than $\sqrt{\texttt{realmax}}$ then x^2 or y^2 will overflow and produce the result `Inf` and hence also $r = $ `Inf`. Consider as an example $x = 1.5e200$ and $y = 3.6e195$. Then

$$r^2 = 2.25e400 + 12.96e390 = 2.250000001296e400 > \texttt{realmax},$$

but $r = 1.500000000432e200$ would well be in the range of the machine numbers. There are remedies to compute r without overflow. One possibility is to factor out:

```
>> x=1.5e200
```

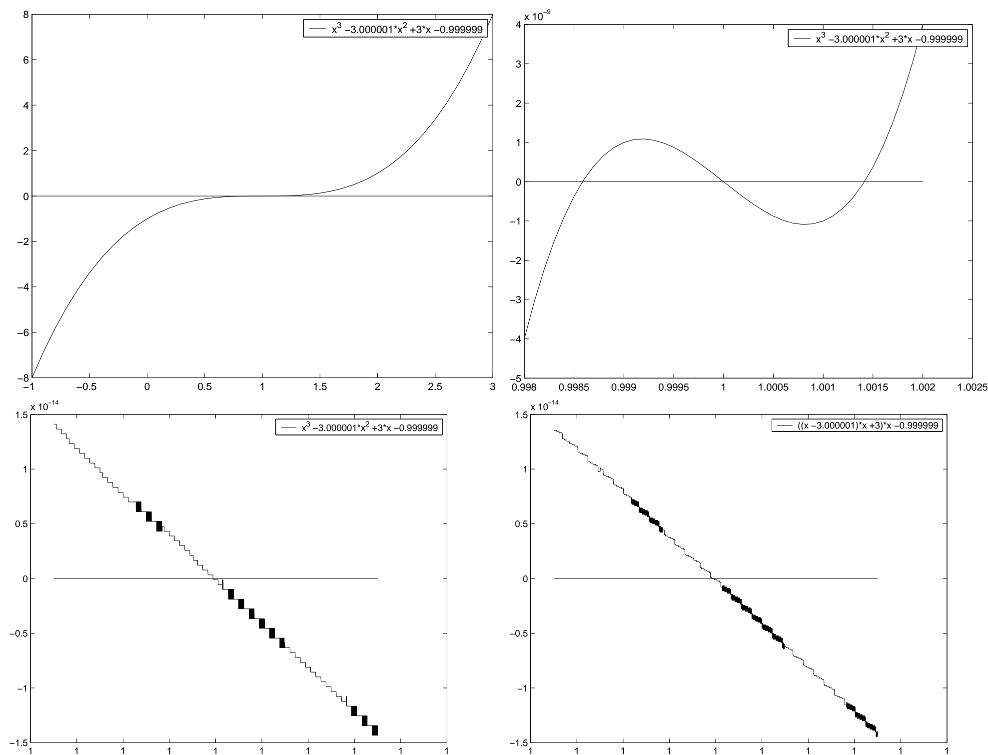

FIGURE 1.3. *Monotonicity is lost*

```
x =     1.500000000000000e+200

>> y=3.6e195
y =     3.600000000000000e+195

>> if abs(x)>abs(y),
       r = abs(x)*sqrt(1+(y/x)^2)
   elseif y==0,
       r = 0
   else
       r = abs(y)*sqrt((x/y)^2+1)
   end

r = 1.500000000432000e+200
```

A simpler program (with more operations) is the following:

```
m = max(abs(x), abs(y));
if m==0, r=0
else r = m*sqrt((x/m)^2+(y/m)^2)
end
```

Note that with both solutions we also avoid possible underflow when computing r.

1.4.7 Test for Overflow

Assume we want to compute x^2 but we need to know if it overflows. With the IEEE Standard it is simple to detect this:

```
if x^2 == Inf
```

Without IEEE the computation might stop with an error message. A machine independent test which will work in almost all cases is

```
if 1/x/x == 0  % then x^2 will overflow
```

To avoid the denormalized numbers the test should be

```
if eps/x/x == 0  % then x^2 will overflow
```

The latter test is almost always correct. In the IEEE Standard `realmin` and `realmax` are not quite symmetric since the equation

$$\texttt{realmax} \times \texttt{realmin} = c \approx 4$$

holds with some constant c which depends on the processor used and/or version of MATLAB.

1.4.8 Cancellation

A special rounding error is called *cancellation*. If we subtract two almost equal numbers, leading digits are canceled. Consider the two numbers with 5 decimal digits:

$$\begin{array}{rcl} 1.2345e0 & & \\ -1.2344e0 & & \\ \hline 0.0001e0 & = & 1.0000e{-4} \end{array}$$

If the two numbers were exact, the result delivered by the computer would also be exact. But if the first two numbers are already obtained by previous calculations and affected by rounding errors then the result is in the best case $1.XXXXe{-4}$ and the digits denoted by X are unknown.

This is exactly what happened in our example at the beginning of this chapter. To compute $\sin(\alpha/2)$ from $\sin\alpha$ we used the formula (Equation 1.1):

$$\sin\frac{\alpha_n}{2} = \sqrt{\frac{1-\sqrt{1-\sin^2\alpha_n}}{2}}.$$

Since $\sin\alpha_n \to 0$ the nominator on the right hand side is

$$1-\sqrt{1-\varepsilon^2}, \quad \text{with small} \quad \varepsilon = \sin\alpha_n$$

and is subjected to severe cancellation. This is the reason why the algorithm performed so badly though theory and program are correct.

It is possible in this case to rearrange the computation and avoid cancellation:

$$\sin\frac{\alpha_n}{2} = \sqrt{\frac{1-\sqrt{1-\sin^2\alpha_n}}{2}} = \sqrt{\frac{1-\sqrt{1-\sin^2\alpha_n}}{2}\,\frac{1+\sqrt{1-\sin^2\alpha_n}}{1+\sqrt{1-\sin^2\alpha_n}}}$$

$$= \sqrt{\frac{1-(1-\sin^2\alpha_n)}{2(1+\sqrt{1-\sin^2\alpha_n})}} = \frac{\sin\alpha_n}{\sqrt{2(1+\sqrt{1-\sin^2\alpha_n})}}$$

The last expression does not suffer anymore from cancellation. The new program becomes:

ALGORITHM 1.3. *Coputation of π, Stable Version*

```
% computation of pi, stabilized version
oldA = 0;
s = sqrt(3)/2; newA=3*s; n=6;     % initialization
z = [ newA-pi n  newA  s];        % store the results
while (newA>oldA)                 % iterate as long as new surface
                                  % is larger than old one
   oldA=newA;
   s= s/sqrt(2*(1+sqrt((1+s)*(1-s))));  % new sin-value
   n=2*n; newA= n/2*s;
   z = [z; newA-pi n newA  s];
end
m = length(z);
for i=1:m
   fprintf('%10d %20.15f %20.15f\n', z(i,2),z(i,3), z(i,1))
end
```

This time we do converge to the correct value of π (see Table 1.2). Notice also the elegant termination criterion: since theoretically the surface of the next polygon grows we have

$$A_6 < \cdots < A_n < A_{2n} < \pi.$$

However, in finite arithmetic this cannot be true forever since there is only a finite set of machine numbers. Thus the situation must occur that $A_{2n} \leq A_n$ and this is the condition to stop the iteration.

Consider as a second example for cancellation the computation of the exponential function using the Taylor series:

$$e^x = \sum_{j=0}^{\infty} \frac{x^j}{j!} = 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} + \ldots$$

It is well known that the series converges for any x. A naive approach is therefore:

ALGORITHM 1.4. *Computation of e^x, Naive Version*

```
function y = e(x, tol);
%
sn = 1; term = 1; k=1;
while abs(term)>tol*abs(sn)
   s = sn; term = term*x/k;
   sn = s + term; k=k+1;
end
y = sn;
```

n	A_n	$A_n - \pi$
6	2.598076211353316	−0.543516442236477
12	3.000000000000000	−0.141592653589793
24	3.105828541230249	−0.035764112359544
48	3.132628613281238	−0.008964040308555
96	3.139350203046867	−0.002242450542926
192	3.141031950890509	−0.000560702699284
384	3.141452472285462	−0.000140181304332
768	3.141557607911857	−0.000035045677936
1536	3.141583892148318	−0.000008761441475
3072	3.141590463228050	−0.000002190361744
6144	3.141592105999271	−0.000000547590522
12288	3.141592516692156	−0.000000136897637
24576	3.141592619365383	−0.000000034224410
49152	3.141592645033690	−0.000000008556103
98304	3.141592651450766	−0.000000002139027
196608	3.141592653055036	−0.000000000534757
393216	3.141592653456104	−0.000000000133690
786432	3.141592653556371	−0.000000000033422
1572864	3.141592653581438	−0.000000000008355
3145728	3.141592653587705	−0.000000000002089
6291456	3.141592653589271	−0.000000000000522
12582912	3.141592653589663	−0.000000000000130
25165824	3.141592653589761	−0.000000000000032
50331648	3.141592653589786	−0.000000000000008
100663296	3.141592653589791	−0.000000000000002
201326592	3.141592653589794	0.000000000000000
402653184	3.141592653589794	0.000000000000001
805306368	3.141592653589794	0.000000000000001

TABLE 1.2. *Stable Computation of* π

For small $|x|$ this program works quite well:

```
>> e(1,1e-8)
ans =      2.718281826198493e+00
>> exp(1)
ans =      2.718281828459045e+00
>> e(-1,1e-8)
ans =      3.678794413212817e-01
>> exp(-1)
ans =      3.678794411714423e-01
```

But for $x = -20$ and $x = -50$ we obtain

```
>> e(-20,1e-8)
ans =      5.621884467407823e-09
>> exp(-20)
ans =      2.061153622438558e-09
>> e(-50,1e-8)
ans =      1.107293340015503e+04
>> exp(-50)
ans =      1.928749847963918e-22
```

completely wrong results. The reason is that e.g. for $x = -20$ the terms of the series

$$1 - \frac{20}{1!} + \frac{20^2}{2!} - \cdots + \frac{20^{20}}{20!} - \frac{20^{21}}{21!} + \cdots$$

become large and have oscillating signs. The largest terms are

$$\frac{20^{19}}{19!} = \frac{20^{20}}{20!} = 4.3e7.$$

The partial sums should converge to $e^{-20} = 2.06e{-9}$. But because of the growth of the terms the partial sums become large as well and oscillate as shown in Figure 1.4. Table 1.3 shows that the largest partial sum has about the same size as the largest term. Since the large partial sums have to be *diminished by additions/subtractions of terms* this cannot happen without cancellation. It also does not help to sum up first all positive and negative parts separately because finally when the two sums are subtracted we suffer again from catastrophic cancellation. Since the result

$$e^{-20} \approx 10^{-17}\frac{20^{20}}{20!}$$

is about 17 decimal digits smaller than the largest intermediate partial sum and the IEEE Standard has only about 16 decimal digits accuracy we cannot expect to obtain one correct digit anymore.

A third example for cancellation is the recursive computation of the *mean* and the *standard deviation* of a sequence of numbers. Given the real numbers $x_1, x_2, \ldots, x_n$ the mean is

$$\mu_n = \frac{1}{n}\sum_{i=1}^{n} x_i. \tag{1.2}$$

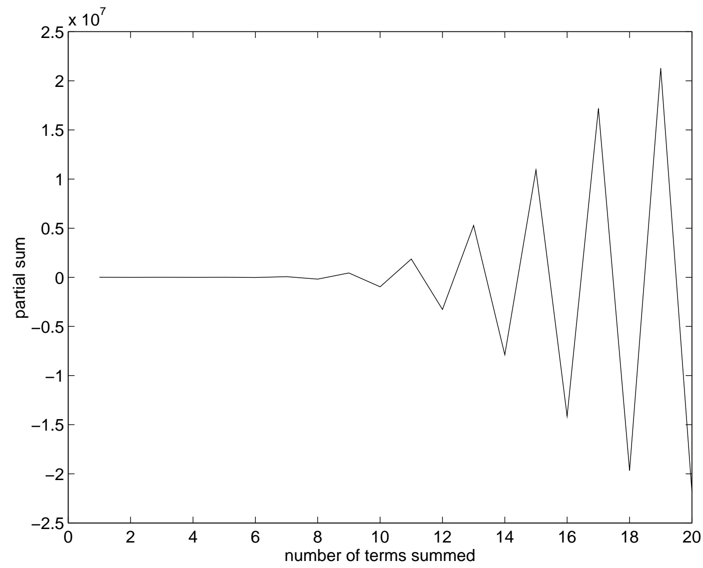

FIGURE 1.4. *Partial sum of the Taylor expansion of* e^{-20}

number of terms summed	partial sum
20	$-2.182259377927747e+07$
40	$-9.033771892137873e+03$
60	$-1.042344520180466e-04$
80	$6.138258384586164e-09$
100	$6.138259738609464e-09$
120	$6.138259738609464e-09$
exact value	2.061153622438558e-09

TABLE 1.3. *Numerically Computed Partial Sums of* e^{-20}

One definition of the variance is

$$\mathrm{var}(\boldsymbol{x}) = \frac{1}{n}\sum_{i=1}^{n}(x_i - \mu_n)^2. \tag{1.3}$$

The square-root of the variance is the standard deviation

$$\sigma_n = \sqrt{\mathrm{var}(\boldsymbol{x})}. \tag{1.4}$$

To compute the variance using Equation (1.3) requires two runs through the data x_i. By the following algebraic manipulation of the formula for the variance we obtain a new expression which allows us to compute both quantities in *only one run through the data*. By expanding the square bracket we obtain from Equation (1.3)

$$\mathrm{var}(\boldsymbol{x}) = \frac{1}{n}\sum_{i=1}^{n}(x_i^2 - 2\mu_n x_i + \mu_n^2) = \frac{1}{n}\sum_{i=1}^{n}x_i^2 - 2\mu_n\frac{1}{n}\sum_{i=1}^{n}x_i + \mu_n^2\frac{1}{n}\sum_{i=1}^{n}1,$$

which simplifies to

$$\sigma_n^2 = \frac{1}{n}\sum_{i=1}^{n}x_i^2 - \mu_n^2. \tag{1.5}$$

This relation leads to the classical recursive computation of mean, variance and standard deviation. In the following test we use the values

```
x = 100*ones(100,1) + 1e-5*(rand(100,1)-0.5)
```

and compare the results with the (two run) MATLAB-functions `mean`, `var` and `std`:

ALGORITHM 1.5.
Mean, Standard Deviation – Classical Unstable Computation

```
x = 100*ones(100,1) + 1e-5*(rand(100,1)-0.5);
format long
s = 0;
sq = 0;
n = 0;
done = 0;
while n<length(x),
  n = n+1;
  s = s+x(n);
  sq = sq + x(n)^2;
  mue = s/n;
  sigma2 = sq/n -mue^2;
  sigma = sqrt(sigma2);
end
means =  [mue mean(x)]
variances =  [sigma2 var(x,1)]
sigma = sqrt(sigma2);
standarddev =  [sigma, std(x,1)]
```

Each execution of these statements will be different since we use the function `rand` to generate the x_i. However, we typically get results like

```
means =
   1.0e+02 *
   1.00000000308131   1.00000000308131
variances =
   1.0e-11 *
   0.90949470177293   0.81380653750974
standarddev =
   1.0e-05 *
   0.30157829858478   0.28527294605513
```

which show that the classical formulas are numerically unstable. It may even occur that the standard deviation becomes complex because the variance becomes negative! Of course this is a numerical effect due to severe cancellation which can occur when using Equation (1.5).

A better update formula which avoids cancellation can be derived as follows:

$$\begin{aligned}
(n+1)\sigma_{n+1}^2 &= \sum_{i=1}^{n+1}(x_i-\mu_{n+1})^2 \\
&= \sum_{i=1}^{n}(x_i-\mu_{n+1})^2 + (x_{n+1}-\mu_{n+1})^2 \\
&= \sum_{i=1}^{n}\left((x_i-\mu_n)-(\mu_{n+1}-\mu_n)\right)^2 + (x_{n+1}-\mu_{n+1})^2 \\
&= \sum_{i=1}^{n}(x_i-\mu_n)^2 - 2(\mu_{n+1}-\mu_n)\sum_{i=1}^{n}(x_i-\mu_n) \\
&\qquad +n(\mu_{n+1}-\mu_n)^2 + (x_{n+1}-\mu_{n+1})^2 \\
&= n\sigma_n^2 + 0 + n(\mu_{n+1}-\mu_n)^2 + (x_{n+1}-\mu_{n+1})^2.
\end{aligned}$$

For the mean we have the relation

$$(n+1)\mu_{n+1} = n\mu_n + x_{n+1}$$

thus

$$\mu_n = \frac{n+1}{n}\mu_{n+1} - \frac{1}{n}x_{n+1}$$

and therefore

$$n\left(\mu_{n+1}-\mu_n\right)^2 = n\left(\mu_{n+1} - \frac{n+1}{n}\mu_{n+1} + \frac{1}{n}x_{n+1}\right)^2 = \frac{1}{n}\left(x_{n+1}-\mu_{n+1}\right)^2.$$

Using this in the recursion for σ_{n+1}^2 we obtain

$$(n+1)\sigma_{n+1}^2 = n\sigma_n^2 + \frac{n+1}{n}\left(x_{n+1}-\mu_{n+1}\right)^2$$

and finally (we set $n := n-1$):

$$\sigma_n^2 = \frac{n-1}{n}\sigma_{n-1}^2 + \frac{1}{n-1}\left(x_n - \mu_n\right)^2 . \tag{1.6}$$

This time we obtain with the same x_i values as above

ALGORITHM 1.6.
Mean, Standard Deviation – Stable Computation

```
x = 100*ones(100,1) + 1e-5*(rand(100,1)-0.5);
s = x(1);
mue = s;
sigma2 = 0;
n = 1;
done = 0;
while n<length(x),
  n= n+1;
  s = s+x(n);
  mue = s/n;
  sigma2 = (n-1)*sigma2/n +(x(n) -mue)^2/(n-1);
  sigma = sqrt(sigma2);
end
means =  [mue mean(x)]
variances =  [sigma2 var(x,1)]
sigma = sqrt(sigma2);
standarddev =  [sigma, std(x,1)]
```

the much better results:

```
means =
   1.0e+02 *
   1.00000000308131   1.00000000308131
variances =
   1.0e-11 *
   0.81380653819342   0.81380653750974
standarddev =
   1.0e-05 *
   0.28527294617496   0.28527294605513
```

1.5 Machine-independent Algorithms

When designing algorithms for finite arithmetic we need to make use of the properties discussed in the previous sections. Such algorithms work *thanks to the rounding errors* and *thanks to the finite set of machine numbers.*

Consider as an example again the computation of the exponential function using the Taylor series. We saw that for $x > 0$ we get good results. Using the *Stirling Formula* $n! \sim \sqrt{2\pi}\left(\frac{n}{e}\right)^n$ we see that for a given x the n-th term

$$t_n = \frac{x^n}{n!} \sim \frac{1}{\sqrt{2\pi}}\left(\frac{xe}{n}\right)^n \to 0, \quad n \to \infty.$$

The largest term is for $n \approx |x|$. After that the terms decrease and converge to zero. Also numerically the term t_n becomes so small that in finite arithmetic

$$s_n + t_n = s_n, \quad \text{with} \quad s_n = \sum_{i=0}^{n} \frac{x^i}{i!}$$

holds. This is an elegant termination criterion which does not depend on the details of the floating point arithmetic but makes use of the finite numbers of digits in the mantissa. This way the algorithm is machine independent; it would not work in exact arithmetic, however, since it would never terminate.

To avoid the cancellation we make use of the property of the exponential function $e^x = 1/e^{-x}$. For $x < 0$ we compute first $e^{|x|}$ and then $e^x = 1/e^{|x|}$. We thus get a stable version to compute the exponential function:

ALGORITHM 1.7. *Stable Computation of* e^x

```
function y = e2(x);
% E2  stable computation of the exponential
%     function using the series.
if x<0, v=-1; x= abs(x); else v=1; end
s = 0; sn = 1; term = 1; k=0;
while s ~= sn
   s = sn;  k=k+1;  term = term*x/k;
   sn = s + term;
end
if v<0, y = 1/sn; else y = sn; end
```

Now we obtain very good results for all x:

```
>> e2(-20)
ans =     2.061153622438558e-09
>> exp(-20)
ans =     2.061153622438558e-09

>> e2(-50)
ans =     1.928749847963917e-22
>> exp(-50)
ans =     1.928749847963918e-22
```

Note that we have to compute the terms recursively

$$t_k = t_{k-1}\frac{x}{k} \quad \text{and not explicitly} \quad t_k = \frac{x^k}{k!}$$

in order to avoid possible overflows in the nominator or denominator.

As a second example consider the problem of designing an algorithm to compute the square root. Given $a > 0$ we wish to compute

$$x = \sqrt{a} \iff f(x) = x^2 - a = 0.$$

Applying Newton's iteration we obtain

$$x - \frac{f(x)}{f'(x)} = x - \frac{x^2 - a}{2x} = \frac{1}{2}(x + \frac{a}{x})$$

and the quadratically convergent iteration (often also called *Heron's formula*)

$$x_{k+1} = (x_k + a/x_k)/2. \tag{1.7}$$

When should we terminate the iteration? We could of course test if successive iterations match to some relative tolerance. But here we can develop a much nicer termination criterion. The geometric interpretation of Newton's method shows us that if $\sqrt{a} < x_k$ then $\sqrt{a} < x_{k+1} < x_k$. Thus if we start the iteration with $\sqrt{a} < x_0$ then the sequence $\{x_k\}$ is *monotonically decreasing* to $s = \sqrt{a}$. This monotonicity cannot hold forever on a machine with finite arithmetic. So when it is lost we have reached machine precision.

We must make sure that $\sqrt{a} < x_0$. But this is easily achieved because it is again geometrically clear that after the first iteration starting with any positive number the next iterate is larger than $\sqrt{a}$. If we start with $x_0 = 1$, the next iterate is $(1 + a)/2 \geq \sqrt{a}$. Thus we obtain Algorithm 1.8.

ALGORITHM 1.8. *Computing $\sqrt{x}$ machine independently*

```
function y = squareroot(a);
%  SQUAREROOT computes y = sqrt(a) using Newton's method
xo =  (1+a)/2; xn = (xo+a/xo)/2;
while xn<xo
  xo = xn;   xn = (xo+a/xo)/2;
end
y=(xo+xn)/2;
```

Notice that Algorithm 1.8 is elegant, there is no "epsilonic" for a termination criterion. It computes the square root on any computer without knowing the machine precision by making use of the fact that there is always only a *finite set of machine numbers.* Finally it is an algorithm that would not work on a machine with exact arithmetic — it does make use of finite arithmetic. Often these are the best algorithms one can design.

Another example of a fool-proof and machine-independent algorithm is given in Chapter **??**. The bisection algorithm to find a simple zero makes use of the fact that there is only a finite number of machine numbers. Bisection is continued as long as there is a machine number in the interval (a, b). When the interval consists only of the endpoints then the iteration is terminated in a machine-independent way. See Algorithm **??** for details.

1.6 Termination Criteria

We have used in the last section termination criteria which were very specific to the problem and which made use of the finite arithmetic. What to do in a general case?

1.6.1 Test Successive Approximations

If we have a sequence x_k converging to some limit s which we are interested in, a commonly used termination criterion is to check the difference of two successive approximations

$$|x_{k+1} - x_k| < \texttt{tol} \quad \text{absolute or} \quad |x_{k+1} - x_k| < \texttt{tol}|x_{k+1}| \quad \text{relative "error"}.$$

The test involves the absolute (or relative) difference of two successive iterates (often referred to somewhat sloppy as absolute or relative error) and of course it is questionable if also the corresponding errors $|x_{k+1} - s|$ and $|x_{k+1} - s|/|s|$ are small. This is certainly not the case if convergence is very slow (see Chapter **??** , Equation (**??**)). We can be far away from the solution s and make very small steps toward it. In that case the above termination criterion will terminate the iteration prematurely.

Consider as an example the equation $xe^{10x} = 0.001$. A fixed point iteration is obtained by adding x an both sides and dividing by$1 + e^{10x}$:

$$x_{k+1} = \frac{0.001 + x_k}{1 + e^{10x_k}}.$$

If we start the iteration with $x_0 = -10$ we obtain the iterates

$$x_1 = -9.9990, \quad x_2 = -9.9980, \quad x_3 = -9.9970.$$

It would be wrong to conclude that we are close to the solution $s \approx -9.99$ since the only solution of this equation is $s = 0.0009901473844$.

1.6.2 Check Residual

Another possibility to check if an approximation is good enough is to check how well it fulfills the property of the object it should approximate. In case of the square root above one might want to check if $r = |x_k^2 - a|$ is small. In case of a system of linear equations $A\boldsymbol{x} = \boldsymbol{b}$ one checks how small the residual

$$\boldsymbol{r} = \boldsymbol{b} - A\boldsymbol{x}_k$$

becomes for an approximative solution $\boldsymbol{x}_k$.

Unfortunately *a small residual does not guarantee that we are close to a solution*! Take as an example for this the following linear system of equations:

$$A\boldsymbol{x} = \boldsymbol{b}, \quad A = \begin{pmatrix} 0.4343 & 0.4340 \\ 0.4340 & 0.4337 \end{pmatrix} \quad \boldsymbol{b} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}.$$

The exact solution is

$$\boldsymbol{x} = \frac{1}{9}\begin{pmatrix} -43370000 \\ 43400000 \end{pmatrix} = \begin{pmatrix} -4.81888\ldots \\ 4.82222\ldots \end{pmatrix} 10^6.$$

The entries of the matrix A are decimal numbers with 4-digits. The best 4-digits decimal approximation to the exact solution is

$$\boldsymbol{x}_4 = \begin{pmatrix} -4819000 \\ 4822000 \end{pmatrix}.$$

Now if we compute the residual of that approximation we obtain:

$$\boldsymbol{r}_4 = \boldsymbol{b} - A\boldsymbol{x}_4 = \begin{pmatrix} 144.7 \\ 144.6 \end{pmatrix}$$

rather a large residual! We can easily guess "better" solutions. If e.g. we propose

$$\boldsymbol{x}_1 = \begin{pmatrix} -1 \\ 1 \end{pmatrix} \quad \Rightarrow \boldsymbol{r}_1 = \boldsymbol{b} - A\boldsymbol{x}_1 = \begin{pmatrix} 1.0003 \\ 0.0003 \end{pmatrix}$$

the residual is much smaller. And the residual of $\boldsymbol{x} = (0,0)^T$ is $\boldsymbol{r} = \boldsymbol{b} = (1,0)^T$, even smaller! *Thus we better not trust small residuals to always imply that we are close to a solution.*

1.7 Condition and Stability

A few words have to be said about *condition* and *stability* though those notions are not part of finite arithmetic. However, they have an important influence in connection with numerical computations.

A problem can be *well-* or *ill-conditioned.* Well-conditioned means that the *solution of a slightly perturbed problem* (this is a problem with slightly changed initial data) *does not differ much from the solution of the original problem.* Ill-conditioned problems are problems where the solution is very sensitive with respect to small changes in the initial data.

A related notion is *well-* or *ill-posed problem.* Let $A : X \to Y$ be a mapping of some space X to Y. The problem $Ax = y$ is well-posed if

1. For each $y \in Y$ there exists a solution $x \in X$.
2. The solution x is unique.
3. The solution x is a continuous function of the the data.

If one of the conditions is not met then the problem is said to be *ill-posed.* Especially if condition 3 is violated then the problem is *ill-conditioned.* But we speak also of an ill-conditioned problem if the problem is well-posed but if the solution is very sensitive with respect to small changes in the date.

A good example of an ill-conditioned problem is the Wilkinson-polynomial discussed in Chapter **??**.

1.8 Principle of Wilkinson

Operations on the computer are subjected to rounding errors. Thus for instance for the multiplication of two numbers

$$a \tilde{\times} b = a \times b(1+r) = a \times (b + b \times r).$$

This means

> *The result of a numerical computation on the computer is the exact result with slightly perturbed initial data.*

The numerical result of the multiplication $a\tilde{\times}b$ is the exact result $a \times \tilde{b}$ with a slightly changed operand $\tilde{b} = b + b \times r$.

The study of the condition of a problem is therefore very important since we will always obtain a solution of a perturbed problem when performing computations with real numbers on a computer.

1.9 The Condition of a System of Linear Equations

Consider the linear system of equations

$$A\boldsymbol{x} = \boldsymbol{b}, \quad \text{with } A \in \mathbb{R}^{n\times n} \text{ nonsingular.}$$

A perturbed system is $(A+E)\boldsymbol{y} = \boldsymbol{b}$. The difference of both equations gives

$$A(\boldsymbol{x}-\boldsymbol{y}) - E\boldsymbol{y} = 0 \iff \boldsymbol{x}-\boldsymbol{y} = A^{-1}E\boldsymbol{y}$$

Taking norms we get

$$\|\boldsymbol{x}-\boldsymbol{y}\| = \|A^{-1}E\boldsymbol{y}\| \le \|A^{-1}\|\|E\|\|\boldsymbol{y}\|$$

Thus if the perturbation is small $\|E\| = \varepsilon\|A\|$ compared to the norm of A then

$$\frac{\|\boldsymbol{x}-\boldsymbol{y}\|}{\|\boldsymbol{y}\|} \le \|A^{-1}\|\|E\| = \varepsilon \underbrace{\|A^{-1}\|\|A\|}_{\kappa(A)}$$

where $\kappa(A) = \|A^{-1}\|\|A\|$ denotes the *condition number*. If we use the 2-norm as matrix norm (see Chapter **??**) then

$$||A|| := \max_{\boldsymbol{x}\neq 0} \frac{||A\boldsymbol{x}||_2}{||\boldsymbol{x}||_2} = \sigma_{\max}(A),$$

and the condition number is computed by

$$\kappa = \frac{\sigma_{\max}(A)}{\sigma_{\min}(A)} = \texttt{cond(A)} \text{ in M}\textsc{atlab}.$$

Thus according to the principle of Wilkinson we have to expect that the numerical solution may deviate by about κ units in the last digit from the exact solution.

1.10 Stable and Unstable Algorithms

If executed in finite arithmetic an algorithm is called stable if the effect of rounding errors is bounded. If an algorithm increases the condition number of a problem then we also classify it as unstable.

EXAMPLE 1.1. *Consider the linear system*

$$A\boldsymbol{x} = \boldsymbol{b}, \quad A = \begin{pmatrix} 10^{-7} & 1 \\ 1 & 1 \end{pmatrix} \quad \boldsymbol{b} = \begin{pmatrix} 1 \\ 2 \end{pmatrix}.$$

If we interpret the equations as lines in the plane then their graphs show a clear cutting point near $\boldsymbol{x} = (1,1)^T$. *The angle between the two lines is about* 45° *degrees.*

If we want to solve the system algebraically then we might eliminate the first unknown in the second equation by replacing the second equation with the linear combination

$$<\textit{second equation}> - \frac{1}{10^{-7}}<\textit{first equation}>.$$

Thus we get the new system

$$\begin{array}{rcrcl} 10^{-7}x_1 & + & x_2 & = & 1 \\ & & (1-10^7)\,x_2 & = & 2-10^7 \end{array}$$

If we again interpret the two equations as lines in the plane then this time we see that the two lines are almost parallel and coincident. The cutting point is not easy to draw – the problem has become very ill-conditioned.

What went wrong? We eliminated the unknown using the pivot in the diagonal which is very small. We transformed a well-conditioned problem in this way into an ill-conditioned one. Choosing small pivots makes Gaussian elimination unstable – we need to apply a pivot strategy *to get a numerically satisfactory algorithm (cf. Section* **??***).*

Note, however, that if we solve linear equations using orthogonal transformations (Givensrotations, or Householder-reflections) then the condition number of the transformed matrices remains constant. This is easy to see. If $Q^TQ = I$ and

$$A\boldsymbol{x} = \boldsymbol{b} \quad \Rightarrow \quad QA\boldsymbol{x} = Q\boldsymbol{b}$$

then $\kappa(QA) = \|(QA)^{-1}\|\|QA\| = \|A^{-1}Q^T\|\|QA\| = \|A^{-1}\|\|A\|$ since the norm is invariant under multiplication with orthogonal matrices.

Example 1.2. *As a second example we consider the computation of the values*

$$\cos(1), \cos(\frac{1}{2}), \cos(\frac{1}{4}), \ldots \; \cos(2^{-12}).$$

We consider three algorithms

1. straightforward:

$$z_k = \cos(2^{-k}), \quad k = 0, 1, \ldots, 12.$$

This is no doubt stable but maybe not efficient.

2. double angle: *we use the relation* $\cos 2\alpha = 2\cos^2\alpha - 1$ *to compute*

$$y_{12} = \cos(2^{-12}), \quad y_{k-1} = 2y_k^2 - 1, \quad k = 12, 11, \ldots, 1.$$

3. half angle: *we use* $\cos\frac{\alpha}{2} = \sqrt{\frac{1+\cos\alpha}{2}}$ *and compute*

$$x_0 = \cos(1), \quad x_{k+1} = \sqrt{\frac{1+x_k}{2}}, \quad k = 0, 1, \ldots, 11.$$

The results are given in Table 1.4. We notice that the y_k *computed by Algorithm 2 are affected by rounding errors while the computations of the* x_k *with Algorithm 3 seems to be stable. Let us analyse the numerical computations of* y_k*. Assume that* y_i

2^{-k}	$y_k - z_k$	$x_k - z_k$
1	-0.0000000005209282	0.0000000000000000
5.000000e-01	-0.0000000001483986	0.0000000000000000
2.500000e-01	-0.0000000000382899	0.0000000000000001
1.250000e-01	-0.0000000000096477	0.0000000000000001
6.250000e-02	-0.0000000000024166	0.0000000000000000
3.125000e-02	-0.0000000000006045	0.0000000000000000
1.562500e-02	-0.0000000000001511	0.0000000000000001
7.812500e-03	-0.0000000000000377	0.0000000000000001
3.906250e-03	-0.0000000000000094	0.0000000000000001
1.953125e-03	-0.0000000000000023	0.0000000000000001
9.765625e-04	-0.0000000000000006	0.0000000000000001
4.882812e-04	-0.0000000000000001	0.0000000000000001
2.441406e-04	0.0000000000000000	0.0000000000000001

TABLE 1.4. *Stable and unstable recursions*

denotes the exact value and $\tilde{y}_i$ *the numerically computed value. The rounding error is the difference* $\varepsilon_i = y_i - \tilde{y}_i$. *Now*

$$\begin{aligned} y_{i-1} &= 2y_i^2 - 1 = 2(\tilde{y}_i + \varepsilon_i)^2 - 1 \\ &= 2\tilde{y}_i^2 + 4\tilde{y}_i\varepsilon_i + 2\varepsilon_i^2 - 1 \\ &= \underbrace{2\tilde{y}_i^2 - 1}_{\tilde{y}_{i-1}} + \underbrace{4\tilde{y}_i\varepsilon_i + 2\varepsilon_i^2}_{\varepsilon_{i-1}} \end{aligned}$$

Thus as first order approximation we get

$$\varepsilon_{i-1} \approx 4\tilde{y}_i\varepsilon_i$$

and therefore

$$\varepsilon_0 \approx 4^i \underbrace{\tilde{y}_1\tilde{y}_2\cdots\tilde{y}_i}_{\approx 1}\varepsilon_i \approx 4^i\varepsilon_i.$$

For $i = 12$ *and the machine precision* $eps = 2.2204e{-16}$ *we obtain* $4^{12}eps = 3.7e{-9}$ *which is a good estimate of the error* $5\mathrm{e}{-10}$ *of* y_0.